

# First Class Refactoring

Adam Pingel  
Department of Computer Science, UCLA  
Pingel at gmail.com

## ABSTRACT

Much of the software engineering paradigm is still based on code as text. De-emphasize the notion of “code as text” in favor of “code as pattern”, along with efficient implementations of the interfaces between the various stages of the software engineering process will enable programmers to interact with the information systems they are creating in more powerful and less error-prone ways. With this goal in mind, this paper argues that refactoring – which is usually considered to be an atomic, nearly instantaneous, ephemeral text manipulation – should be thought of as consisting of objects or processes available for inspection. Several specific changes to the Eclipse Refactoring API are proposed. Initial work on the first proposal is described.

## 1. INTRODUCTION

Consider the various processes of software engineering. The basic story starts with a collection of text files. The text files are fed into a compiler that turns the text into a token stream. The tokens are parsed into an abstract syntax tree (AST). Flow analysis is performed on the AST to analyze the type soundness of the code. Eventually registers are allocated, loops are unrolled, and the byte code is produced.

Expand the environment a bit to include more of the context. Programmers make changes to the source code and check these changes into the source code control system. Style checkers ensure that the source code conforms to the proper style for the given engineering environment. Test cases are run in order to check that important behaviors have not been disturbed by the changes. Model checkers can prove that a set of invariants is maintained, or provide counter-examples if they are not. Profilers run to help engineers understand how resources like CPU cycles and memory are consumed throughout the execution of the code.

These steps occur in large blocks. Once a change to the source code is made, each step in this process must be applied in sequence to the result.

It is certainly the case that this pipeline of processes can be automated. But in the real world, many of these steps are considered optional. And frequently they are only applied at important milestones.

This paper proposes that all these steps should be performed incrementally as small changes are committed to the source code. This emphasis on applying as much of

the software engineering pipeline incrementally is similar to the Extreme Programming philosophy of frequent refactoring and testing. How can the steps of the software engineering process be modified to reuse calculation in order to support incremental changes?

## 2. THE CURRENT STATE OF AFFAIRS

### 2.1 Refactoring

The paper is concerned with a class of changes to source code called "refactoring". Refactoring is commonly defined as "behavior preserving transformations". One simple way to understand them is that they are fast, error-free ways of applying changes to text. Many such patterns of changes occur with high frequency. These have been named and defined in such works as Martin Fowler's seminal opus, "Refactoring".

The simplest example of a refactoring is Rename Variable, which does exactly what it sounds like. A more complex example is Inline Method, which replaces method calls with the body of the method. Refactorings are most commonly thought of with regard to object-oriented languages, so they include patterns such as Push Down Method and Extract Interface. Some are fairly sophisticated, as exemplified by the Replace Conditional with Polymorphism.

### 2.2 Extract Interface

Initial work for this project focuses on the Extract Interface refactoring. One motivating example involves extracting the externally observable behavior of a state manager in order to support concrete and proxy implementations.

Consider a Bank class that manages the state of a bank, which for purposes of this exposition will require just four methods:

```
class Bank {
    void deposit(int accountNo, int amount) { ... }
    boolean withdrawal(int accountNo, int amount) { ... }
    int balance(int accountNo) { ... }
    void commit(OutputStream out) { ... }
}
```

The first three methods have the obvious meaning. The last one is intended to define a mechanism for implementing persistence in this simple state manager. Now let's add a little code that uses the Bank class:

```

class BankUtil {
    void payroll(Bank bank) {
        bank.deposit(1, 3000);
        bank.deposit(2, 2800);
    }

    void clear(Bank bank) {
        bank.withdrawal(1, bank.balance(1));
    }
}

```

Let's say that this code is running fine on a single machine inside a web application. Over time it may become desirable to run the BankUtil class on a different machine than the Bank class. One solution is to create a new interface – IBank – that includes all the methods used by BankUtil. The existing Bank class extends this new interface. A new BankProxy class is created that implements the IBank interface. The code should look something like this:

```

interface IBank {
    void deposit(int accountNo, int amount);
    boolean withdrawal(int accountNo, int amount);
    int balance(int accountNo);
}

class Bank implements IBank {
    void deposit(int accountNo, int amount) { ... }
    boolean withdrawal(int accountNo, int amount) { ... }
    int balance(int accountNo) { ... }
    void commit(OutputStream out) { ... }
}

class BankProxy implements IBank {
    void deposit(int accountNo, int amount) { ... }
    boolean withdrawal(int accountNo, int amount) { ... }
    int balance(int accountNo) { ... }
}

class BankUtil {
    void payroll(IBank bank) {
        bank.deposit(1, 3000);
        bank.deposit(2, 2800);
    }
    void clear(IBank bank) {
        bank.withdrawal(1, bank.balance(1));
    }
}

```

Note that the references to Bank in BankUtil have been changed to IBank in order to achieve the goal of code reuse. At this point, the programmer could add code to BankProxy that knows how to relay the calls across the network.

This is standard object-oriented practice. Extract Interface does nothing magical. It simply bundles this tedious and error-prone set of text changes into a single action.

## 2.2 Eclipse refactoring API

Eclipse 3.0.1's `ExtractInterfaceRefactoring` class lives in the `org.eclipse.jdt.internal.corext.refactoring.structure` package. It extends the abstract `Refactoring` class, which lives in the `org.eclipse.ltk.core.refactoring` package. The public non-final abstract methods of `Refactoring` are:

```
String getName();
RefactoringStatus checkAllConditions(IprogressMonitor pm);
RefactoringStatus checkInitialConditions(IprogressMonitor pm);
RefactoringStatus checkFinalConditions(IprogressMonitor pm);
Change createChange(IprogressMonitor pm);
```

(Note: this list omits information about thrown exceptions.)

A `RefactoringStatus` represents the outcome of a condition checking operation. Its `getSeverity` method returns one of `OK`, `INFO`, `WARNING`, `ERROR`, or `FATAL`.

After a `Refactoring` object, `ref`, is created and all of the appropriate fields are set, it is used to change the code as follows. This code assumes the context of a test case:

```
IUndoManager undoManager = RefactoringCore.getUndoManager();
undoManager.flush();
CreateChangeOperation create =
    new CreateChangeOperation(
        new CheckConditionsOperation(
            ref, CheckConditionsOperation.ALL_CONDITIONS),
        RefactoringStatus.FATAL);
PerformChangeOperation perform =
    new PerformChangeOperation(create);
perform.setUndoManager(undoManager, ref.getName());
ResourcesPlugin.getWorkspace().run(perform,
    new NullProgressMonitor());
RefactoringStatus status = create.getConditionCheckingStatus();
```

The `CheckConditionsOperation` constructor takes a `Refactoring` as its first argument. The `Refactoring` is saved in a private field called `fRefactoring`. The `CheckConditionsOperation` is then passed as the first argument to a `CreateChangeOperation`. The `fRefactoring` is copied from the `CheckConditionsOperation` into a field of the same name. The `CreateChangeOperation` is then passed as the argument to construct a new `PerformChangeOperation`, which saves the `CreateChangeOperation` in the `op` field.

At this point `perform.op.fRefactoring == ref`.

The call to `ResourcesPlugin.getWorkspace().run` ultimately calls the `Refactoring`'s `createChange` method and calls `initializeValidationData` on the result as well as calling `checkAllConditions`, `checkInitialConditions`, and `checkFinalConditions`. It's a complex callback scheme, but in the end the `Refactoring` does all the interesting work in the proper sequence.

### 3. PROPOSED CHANGES

#### 3.1 Continuous refactorings

It is clear from the example code above that Eclipse Refactorings are atomic operations. The first proposal of this project is to explore ways of turning them into ongoing threads or processes with interstitial checkpoints. A motivating example for this arises with the Extract Interface refactoring.

Referencing the Bank example above, the stage can be set for one such enhancement. During the original application of the refactoring there may be references to Bank that cannot be changed to IBank because of a call to the commit method. If at some point these calls are removed, there is no way for the original refactoring to notice this because it was an ephemeral operation.

If the original refactoring were to linger long enough to keep an eye on all of the original call sites, it could continue to propagate these changes as they become possible.

Another change that may occur after the original refactoring is the addition of an existing class method to the interface. If any class references were originally prohibited from changing to interface because of a reference to that method, they should be updated to interface references.

These two kinds of changes are handled properly by an extension implemented as the first piece of this project. It is implemented as a Thread spawned by ExtractInterfaceUtil.updateReferences. This method is called via ExtractInterfaceRefactoring.createChangeManager, which is one of the consequences of Workspace.run.

At present the extension is only able to retain a list of compilation units (roughly “files”) that contains class references. Future implementations should retain a list of individual class references.

A similar idea is to keep a list of pending refactorings. Many refactorings – such as Inline Method – have non-trivial initial conditions. If they are not originally met, it could be advantageous to retain the attempt in a collection of refactorings. If at any time in the future the initial conditions are met, the refactoring could be applied.

#### 3.2 User visibility

Many of the behaviors described in the previous proposal may be undesirable if fully automated and invisible to users. A number of user interface elements and queries should be implemented in order to provide users with complete understanding of all continuous and pending refactorings. This should include the ability to modify, add, and delete refactorings on the list of pending refactorings.

For currently running continuous refactorings, all available details should be available to the UI.

This body of continuous refactorings should also be able to help answer queries like “What are the effects of removing method  $m$  from class  $C$ ?” Obviously there are some aspects of the compiler that are involved in answering this question. The effects of the continuous refactorings should augment the list of effects discovered by the compiler.

### 3.3 Generalized reversibility

Support for “undo refactoring” is alluded to in the example code above. The `undoManager` allows the changes from the refactoring to be rolled back. A complete survey of the refactoring code has not been done, but there is anecdotal evidence that undo support is not widely implemented correctly. Many of the test cases for refactoring include a check for the presence of an undo operation, but do not validate the results of applying the undo operation.

One example of unimplemented undo support comes from the Extract Interface. Undoing such a refactoring correctly drops the “implements I” form the class declaration, but does not revert interface references back to class references, nor does it delete the newly created interface.

Why is undo support not automatically achieved? It should be possible to define refactorings in such a way to get this behavior automatically. A strategy for getting there is to list a few cases where undo does not work, implement undo with the current API, define a protocol for uniform undo support, and ultimately make observance of this protocol required as by baking it into the existing refactoring API.

### 3.4 Composite refactorings

Composite refactorings is often useful. Using the Bank example above, it may be the case that the programmer would like to use “Bank” as the name of the interface and change the “Bank” class to “BankImpl”. This could be achieved by following the Extract Interface with a Rename Class and a Rename Interface refactoring.

Another simple example is the definition of Swap Variable Names, which could be implemented as three Rename Variable refactorings executed in sequence.

This enhancement would require new user interface elements and data structures.

### 3.5 Refactoring application tree

Throughout the evolution of a body of software, a number of refactorings may be applied. Some paths may be explored and abandoned. If the set of changes considered is limited to atomic refactorings (which is unusual but not entirely unrealistic for some modes of software engineering) this evolution can be modeled as a tree with nodes representing the state of a project and the arcs representing specific applications of a refactoring.

One way to represent the tree is to record a copy of all code at each node. Is it possible to avoid these copies and store only information about the refactorings? Is it possible to support fast rendering of each of the nodes in the tree?

More interestingly, is it possible to support write-through versions of this tree? The motivation for this question is that it often arises that one programmer wishes to apply a refactoring – like Push Down Method – before another programmer is ready to think of the code in those terms. For a period of time, both views may be supportable.

Example: Consider a class hierarchy with root class A and subclasses B and C. A implements method m. Programmer 1 decides to Push Down Method 'm'. Yet Programmer 2 still desires to see method m as a member of class A.

If we prepend the method with 1 or 2 to denote which programmer's view is referenced, the following notation represents the constraint:

$$1.B.m == 1.C.m == 2.A.m$$

In order to support consistent views for both programmers, any changes to m must also be copied to the other two versions of m. If Programmer 1 decides to make B.m and C.m different, he or she should be notified that doing so would invalidate Programmer 2's view.

What are the limits of supporting views? What operations on one view would invalidate others? When would constraints more complex than equality constraints be useful? These are likely to be difficult questions to answer.

### 3.6 Support for incremental changes

The implementation of the enhancement in 3.1 is very wasteful. The thread that is spawned polls to check for updated modification times of the files of interest. If it notices that a file of interest has been modified, it searches through the corresponding code for references to the original class.

To perform the search, a pattern object is created:

```
SearchPattern pattern=  
    SearchPattern.createPattern(theType,  
                               IJavaSearchConstants.REFERENCES);
```

Where theType is an object that corresponds to the class being searched for.

Other examples of pattern types are DECLARATIONS, IMPLEMENTORS, and WRITE\_ACCESSES. After the pattern object is created, it is passed to the RefactoringSearchEngine.search method.

This search does not reuse computation from any previous searches. In large bodies of code, this could be an inhibiting bottleneck. Is it possible to restructure Eclipse in order to support ongoing, event-driven searches? In that model, the pattern would be created once. Any changes to the code would be evaluated by the pattern and made available to any interested parties. This would likely be useful in contexts other than refactoring.

### 3.7 Refactoring other structures

As noted in the introduction, not all of the structures of interest to software engineers are expressed as java code. How far can refactorings extend beyond the limits of the java project?

Test cases expressed as JUnit TestCases will automatically be included in refactorings and require no special treatment.

Model checkers like BLAST create a representation of the code as a Control Flow Automaton (CFA). A proposal for extending BLAST to work with Java requires that the dynamic method dispatch be implemented as new guards and branches in the CFA. Modeling exceptions requires similar translation. Is it possible to update the CFA incrementally in response to refactorings?

It is interesting to note that a refactoring such as Replace Conditional with Polymorphism would result in an isomorphic CFA.

Some representations of symbolic execution traces model the scope of variables as methods are called. If a method is inlined, the shape of the corresponding pieces of any symbolic execution traces should change as well.

Other examples of data structures of interest include invariants, predicates, observation structures, and correctness proofs. It is theoretically possible to create a refactoring in these domains for any sound Java refactoring.

## 4. LONG-TERM VISION

Some of the seven proposals in the previous section are attainable with concerted but relatively non-invasive programming effort. Others may require very deep rethinking of the Eclipse API. Attempting to redefine the core Eclipse API's while dealing with the complexities of Java may conflate the issues. A strategy that searches for solutions for these seven proposals in a much simpler object-oriented language may turn out to be shortest path to implementing these proposals.

The precise fruits of this investigation aren't known, but some of them suggest provocative analogies. It has been proven for combinatory logic that a system of rewrite rules as small as the traditional combinators S and K is Turing Complete. A refactoring can be thought of as a rewrite rule. Using this perspective, the refactoring tree proposed in 3.5 becomes much like the CFA used to model program flow, with the programmer playing the role of the Turing Machine. The analogy is not precise: computers apply rewrite rules according to much more rigid rules than programmers apply refactorings. Nevertheless it reveals some interesting shared dynamics.

The question of when to apply refactorings has always been difficult to answer. Martin Fowler fatuously resorted to the phrase "when it smells bad" in his book "Refactoring". Recently there has been some work to try to formalize these "bad smells" (see Gene Garcia's <http://wiki.java.net/bin/view/People/SmellsToRefactorings>). Could these be used as heuristics for automatically guided design? If so, what dynamics would these heuristics really be modeling? To what extent are they a good model of the human ability to read and write programs?

In terms of the refactoring tree, a "smelliness" could be computed for each of the nodes. This could be the input to a search algorithm for walking the graph until a node of low smell is encountered. What do these landscapes look like? Is there typically a large basin of attraction? How sensitive is the topology to the specifics of the physics of "smell" and to the particular set of refactorings employed?

There are uncountably infinite variants of programs that implement the same behavior. Programming design concerns go well beyond the correctness of the algorithm. The choice of which representation to pick has as least as much to do with communicating to one's self and one's human peers than it has to do with communicating with a piece of silicon.

Pushing on tools that support refactoring will be useful in its own right, but may also help answer to some of these larger questions.

## 5. ACKNOWLEDGEMENTS

Thanks to Jim Clune, who was my partner in a project for a class taken concurrently in which we implemented a prototype of "Continuous Refactoring" in Eclipse described in section 3.1.